Marcia Gallant

Assignment 4 Justification

Instructions to run the program:

In the test package there is a file called ZombieTrackerTest. That has a Junit class and have a test() method. In order for it to run, you must configure the path and add an external jar file: assertj-core—3.17.2 which is located with the files I submitted.

Composite Design Pattern

I used the design pattern composite since towns consist of zombies and hordes and hordes consist of zombies. This enhances my design because we can keep all the zombies and hordes in the town together in a single object and we can iterate over them all and call the same function to be able to find out how many zombies there are in the town. This makes it much easier because we can count zombies and keep track of how many there are in one object and use the same method. Otherwise, we would need to call instance of and see what object it is and then call the appropriate method for that object to count the zombies. This would make our program hard to extend if we did it like that. Using the composite design pattern makes it easier to design and helps us follow the Liskov substitution principle and the open closed principle.

Observer Design Pattern

I used the observer design pattern to let all the different objects that are in my composite design pattern (zombies, hordes, towns) to update them all when a zombie is killed. This way when someone kills a zombie we do not have to go over all of the object types and look to see if the zombie is in which horde or if it is just a solo zombie in the town and delete it. Instead, it notifies all objects about which zombie is killed and they all update themselves and delete the zombie that was killed if it existed in the object. This way all the objects stay up to date with state changes in the system.

Open Closed principle

I got rid of the hordeInfo from the Zombie class and made an abstract class Horde and 3 subclasses of different types of hordes. In the ZombieTracker class I was violating the open closed principle by having to make new methods for all the different horde information there could be. This makes it hard to extend because there could be a lot of different things you want to store about a horde and it is hard to store that information in one String and having to keep make new methods every time you want to add something into it. Now, you can store whatever new information you want by simply putting into the existing horde sub/class or creating a new horde subclass which means it resolves the violation.

I made an abstract Guild class and a subclass SackvilleGuild. I also added GuildMember class. In the ZombieTracker class I was violating the open closed principle because if I extended the program to more guilds then I would either need to add if statements or more methods for killing a Zombie. This makes it difficult to extend. Now, the guilds consist of GuildMembers and every GuildMember has a method to kill the zombie. This makes it much easier to kill the zombie and we can easily extend it to find out which guild the person was from. This means we can easily extend the program by adding new guilds and guildmembers without adding if statements or new methods in the main class which means it resolves the violation.

Single Responsibility principle

I also made an abstract Town class and Sackville class which extends Town. Town extends CountryComposite. This is so Town can hold all the different zombies and hordes that are in it. This separates the responsibility of knowing what zombies and hordes each town has from the ZombieTracker class since each Town subclass will be responsible for that now. It was violating SRP because ZombieTracker is just suppose to be testing the program, it is not suppose to have to keep track of information for the program to work.